Knapsack Problem

```
def knapsack(weights, values, capacity):
    n = len(weights)  # Number of items
    # Create a 2D DP array where dp[i][w] represents the maximum value that can be achieved with
    # the first i items and a weight limit of w
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Build the DP table
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    # The maximum value will be in dp[n][capacity]
    return dp[n][capacity]

# Example usage:
weights = [2, 3, 4, 5]  # Weights of the items
values = [3, 4, 5, 6]   # Values of the items
capacity = 5            # Capacity of the knapsack

result = knapsack(weights, values, capacity)
print(f"The maximum value that can be carried in the knapsack is: {result}")
```

KRANTI GAIKWAD
14:15 Today

Time complexity: O(n×W)
Space complexity: O(n×W)

```
The maximum value that can be carried in the knapsack is: 7
```

Travelling Salesman Problem

```
import sys

def tsp(graph):
    n = len(graph)
    # dp[mask][i] will hold the minimum cost to visit the set of cities represented by mask
    # and end at city i.
    dp = [[sys.maxsize] * n for _ in range(1 << n)]
    # Starting at city 0, so dp[1][0] = 0 (we're at the starting city with only city 0 visited
    dp[1][0] = 0

    # Iterate over all subsets of cities
    for mask in range(1 << n):
        for u in range(n):
            # If city u is not in the subset represented by mask, continue
            if not (mask & (1 << u)):
                continue
            # Try to go from city u to another city v
            for v in range(n):
                if (mask & (1 << v)) == 0:  # If city v is not in the subset
                    new_mask = mask | (1 << v)
                    dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v])

    # Reconstruct the minimum tour cost
    min_cost = sys.maxsize
    # Now we need to return to the starting city 0 from any city
    for i in range(1, n):
        min_cost = min(min_cost, dp[(1 << n) - 1][i] + graph[i][0])

    return min_cost

# Example usage:
graph = [
    [0, 10, 15, 20, 25],
    [10, 0, 35, 25, 30],
    [15, 35, 0, 30, 5],
    [20, 25, 30, 0, 10],
    [25, 30, 5, 10, 0]
]

# The graph is a 2D matrix where graph[i][j] represents the cost from city i to city j
result = tsp(graph)
print(f"The minimum cost of the trip is: {result}")
```

KRANTI GAIKWAD
14:16 Today

Time complexity: O(n^2×2^n)
Space complexity: O(n×2^n)

```
The minimum cost of the trip is: 65
```

Queen Placement Problem

KRANTI GAIKWAD
14:19 Today

Time complexity: O(N^3)
Space complexity: O(N^2)

```
def is_safe(board, row, col, N):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True


def solve_n_queens_util(board, col, N):
    # If all queens are placed then return True
    if col >= N:
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(N):
        if is_safe(board, i, col, N):
            # Place the queen on the board
            board[i][col] = 1

            # Recur to place the rest of the queens
            if solve_n_queens_util(board, col + 1, N):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution,
            # then backtrack
            board[i][col] = 0

    # If the queen cannot be placed in any row in this column, return False
    return False


def solve_n_queens(N):
    # Initialize the board
    board = [[0 for _ in range(N)] for _ in range(N)]

    # Solve the problem using backtracking
    if not solve_n_queens_util(board, 0, N):
        print("Solution does not exist")
        return

    # Print the solution
    for row in board:
        print(" ".join("Q" if x == 1 else "." for x in row))


# Example usage:
N = 8  # Set the value of N
solve_n_queens(N)
```

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```